Modelling Distributed Erlang within a Single Node

Stavros Aronis
Erlang Solutions AB
Stockholm, Sweden
stavros.aronis@erlang-solutions.com

Viktória Fördős
Klarna Bank AB
Stockholm, Sweden
viktoria.fordos@klarna.com

Dániel Szoboszlay
Klarna Bank AB
Stockholm, Sweden
daniel.szoboszlay@klarna.com

Abstract

This paper was motivated by a challenge we faced while re-architecting a critical component in Klarna's software stack. We wanted to increase our confidence about correctness aspects of a new distributed algorithm, developed for an Erlang system at the very core of Klarna's business. Reasoning about the correctness of concurrent Erlang systems is a difficult task, but tools exist that can help, for instance, Concuerror. However, our algorithm was intimately linked to distributed Erlang's behaviours, which are not supported by Concuerror. The solution we came up with was to design and implement `vnet`, a modelling library which can be used to simulate the behaviour of distributed Erlang nodes within a single Erlang node. We discuss aspects of `vnet` showing its capabilities and limitations. We also report on two case studies, showing how `vnet` can be used to prototype, test and verify simple and advanced distributed Erlang systems. We finally demonstrate that we were able to find errors and verify properties in the systems of our case studies, using Concuerror.

CCS Concepts • **Computing methodologies → Model verification and validation**; • **Software and its engineering → Software verification**;

Keywords distributed Erlang model, Concuerror, software correctness, testing

ACM Reference Format:
Stavros Aronis, Viktória Fördős, and Dániel Szoboszlay. 2018. Modelling Distributed Erlang within a Single Node. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang (Erlang '18), September 29, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3239332.3242764

Erlang '18, September 29, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5824-8/18/09…$15.00
https://doi.org/10.1145/3239332.3242764

1 Introduction

One of the many industrial systems built with Erlang/OTP is KRED, which is the original engine that runs all of Klarna's services. KRED is a distributed system that provides Klarna with a platform for receiving and processing transactions. Even though KRED has been battle tested through the years and is fairly bullet proof, Klarna's engineers have been continuously reviewing core parts of the system year by year.

KRED consists of a number of Erlang nodes which elect a leader (master) node that the rest of the cluster follows. The underlying leader election algorithm is an in-house solution, implemented purely in Erlang, without using any external services. Roles are decided and communicated among the nodes using Erlang messages. The database system of KRED (called KDB) can then depend on the leader election to ensure *consistency*, by performing write transactions on the master node (who acts as what is commonly known as the 'single source of truth') and *availability*, by then replicating those transactions to the follower nodes (keeping their order intact), also using an in-house replication algorithm implemented in Erlang.

Recently, we decided to review the design around the leader election and replication algorithms. While reviewing these aspects we concluded that a redesign was possible that could improve the system's reliability, increase the utilization of KRED nodes and decrease operational complexity. However, given that the scope of the redesign was major, it was essential to gain confidence that the new architecture was correct before approving such a change. We decided, therefore, to perform early prototyping, testing and possibly verification on a model of the system to assess if the new architecture could satisfy a number of requirements. It is easy to see that to get useful information we would have to model a distributed system. As an example, consider the following requirement: the cluster shall always converge to a stable state where there is exactly one master node and the rest of the cluster knows of and follows that node. This is a requirement for the entire distributed system and cannot be checked in a cluster that consists of a single node. Moreover, even though Erlang's concurrency model is in general 'agnostic' of distributed nodes, the behaviour of a distributed system is different than a single-node system, and difficult to test with pure Erlang tools. For example, distributed Erlang's behaviour can cause additional race conditions, as e.g., network is not always stable, so observing and testing

the behaviour of the system when nodes disconnect is also important.

As a result, modelling and testing a single KRED node would not be enough. Additionally, we really wanted to understand how the system behaves when injecting failures on the network. Even though Erlang provides built-in support for distribution, there are no tools that can be used to study how distributed Erlang systems behave, or how race conditions can play out inside a cluster. However, by modelling a distributed Erlang system in a single node, we could use a tool such as Concuerror.

In this paper we present a general model for distributed Erlang systems (distributed Erlang model) that is executable in a single node (Section 3). We then demonstrate how distributed Erlang systems can be implemented using the model (Section 4), by describing two case studies: a model for a simple counter server and a model of the new KRED/KDB design (KRED/KDB model). The implementation of our distributed Erlang model, the `vnet` library, together with the counter server's model, are provided as open source software. Finally we evaluate our work, presenting a number of scenarios and properties that we were able to test and verify on the case studies, using Concuerror (Section 5).

2 Background

We first give a short background on distributed Erlang, in order to identify the key properties that our distributed Erlang model should accommodate. We also briefly discuss stateless model checking and Concuerror – the technique and tool we used to evaluate the models which we build using the distributed Erlang model as case studies.

We assume familiarity with the following basic, single-node concepts: processes, process identifiers (PIDs), message-passing primitive operations (send & receive), process registry, group leaders, and exit signals and all operations involving them (link, monitor, etc.). More details about these concepts can be found in books [14] or other publications [6].

2.1 Distributed Erlang Systems

The basic running environment of an Erlang program is a node. A distributed Erlang system consists of a number of nodes which are connected over a network. Any two nodes can communicate by establishing a direct network connection. In standard setups, every node is connected to every other node, forming a fully connected mesh topology.

Network failures may break some of the connections, or even split the network to isolated islands. Such failures can be detected by processes, using a node-wide monitoring mechanism. This mechanism uses 'ping'-like messages to determine connection status; as a result a network glitch will look the same as a node/process crash. When a connection between nodes fails, any interested process will be notified about the failure with an Erlang message.

An important aspect of Erlang is that messages sent from a process to another will be seen by the other process in the same order. However, in the Erlang/OTP implementation it is always the case that messages to processes in the same node are delivered before the call to the respective send operation is completed. Consider as an example that there are three processes: A, B and C. A sends the message m_1 to B then it sends m_2 to C. When C receives m_2, it sends the message m_3 to B. If all three processes are running in a single node then the message ordering will always be that m_1 arrives before m_3 in B, as message passing never fails within a node. The message cannot be in-flight, so m_1 is placed in the mailbox before the send of m_2 happens, and thus m_3 will always come later. In the distributed setting however, when A, B and C might be running in three different nodes, m_1 might e.g., get delayed on a network socket while m_2 and m_3 go through quickly. In this case we can also observe the reverse ordering, m_3 followed by m_1. This behaviour has motivated the use of different models when reasoning about distributed Erlang systems [10].

We also remind that any process can be contacted using its PID, even if it is running in a different node. Using registered process names in a distributed setup requires more attention, since the name alone cannot uniquely identify the process in the distributed system. Registered names are local, so registering a process with name f in all the nodes of a cluster is possible. To send a message to a registered process running in a different node, the name of the node should also be specified.

2.2 Stateless Model Checking & Concuerror

It is well-known that concurrent programs can behave differently each time they are executed, even when all the inputs remain unchanged, purely due to the non-deterministic nature of the scheduling of their processes, and the various race conditions which exist when these processes access and modify shared data. Verifying that the behaviour of such a program is correct, regardless of the particular scheduling is a hard task.

Model checking is, in general, a technique that can be used to prove properties of any system by exploring the space of all its possible states. When applied to software, however, it is hard to capture and succinctly store the required state information. *Stateless model checking* [12] solves both these problems by summarizing the state of a concurrent program with a *scheduling* of the processes, given as a sequence of execution steps. Only operations that access/modify potentially shared data need to be included in this scheduling. It is then assumed that if the same scheduling is replayed from the initial state, an identical program state will be reached. By using an algorithmically controlled scheduler, one can explore all possible such schedulings, verifying that a program always behaves correctly. This would normally require a vast number of executions, but partial order reduction

techniques [11] can be used to reduce this number, by only exploring schedulings whose racing operations form different partial orderings and proving that any other scheduling is equivalent to those.

While model checking's primary use is safety testing ('something' can never happen), it is of course possible to also use the technique for reachability testing ('something' *can* happen, e.g., by asserting that 'something' is impossible and expecting this assertion to fail).

Concuerror [3, 9, 13] is an open source stateless model checking tool for Erlang programs, that uses an optimal dynamic partial order reduction (DPOR) algorithm [1] and precise information about Erlang operations that may interfere [6], including optimizations for treating Erlang's message-passing-related race conditions [5]. The tool control the scheduling of the processes by performing a code rewrite that inserts instrumentation around racing operations. Concuerror supports the complete Erlang language and can instrument and test programs of any size, automatically including any libraries they use. One limitation, shared by most stateless model checking tools, is that the test must be 'finite', in the sense that it cannot have infinite execution sequences. A more significant limitation is that Concuerror does not currently support multi-node tests, as it is not able to orchestrate and coordinate operations in multiple Erlang nodes, hence tests are restricted to involve only a single Erlang node.

Given a program within these limitations, Concuerror will (by default) explore all (distinct as described before) possible schedulings of the program, until it finds a process crashing or a deadlock (all processes blocked at receive statements). If no such errors are detected, then it is guaranteed that no concurrency errors exist. In this way, the tool can *verify* concurrent Erlang programs.

3 Modelling a Distributed Erlang System

In this section we introduce our distributed Erlang model and present its implementation, the `vnet` library. The source code with an open source license is available at:

https://github.com/klarna/vnet

3.1 Design Considerations

A model should ideally be completely transparent to code running on top of it. When it comes to Erlang, language elements, BIFs, and library modules should work in exactly the same way as in a native distributed Erlang system. While it is acceptable to e.g., replace modules from the Erlang/OTP implementation, such as the one providing remote procedure calls (`rpc`), with alternative implementations that have the same API, redefining BIFs and language elements such as the send operator (`!`) is not desirable.

This naturally leads to two alternatives for creating a virtual network of nodes: either all code running in a virtual node shall refrain from using the real distribution primitives (BIFs and language elements) and use the modelling library's implementation from a custom module instead (such as writing `vnet:send(Pid, Msg)` instead of `Pid ! Msg`), or the library shall work with the native primitives without redefining their behaviour. The `vnet` library uses both approaches: models can use certain native operations, but have to replace others with their `vnet` implementation.

As our ultimate goal is to test and verify the correctness of proof-of-concept implementations of algorithms that can later be turned into final, production-ready versions, we wanted the gap between the model and the final implementation to be as small as possible. As a result, our primary guideline for which primitives to use in their native form and which to replace with function calls to custom modules was whether the primitive is principally needed by standard Erlang/OTP behaviours. We summarize these decisions in Table 1.

Table 1. Modelling Distribution Primitives in `vnet`

Primitive	Modelling approach
Message passing	Native
Linking	Native
Monitoring	Native
Spawning local processes	Native
`erlang:node/0`	`vnet:vnode/0`
`erlang:node/1`	`vnet:vnode/1`
`rpc:call/4`	`vnet:rpc/4`
`rpc:multicall/4`	`vnet:rpc/4`
`global:register_name/2`	`vnet:register_name/2`
`global:unregister_name/1`	`vnet:unregister_name/1`
`global:whereis_name/1`	`vnet:whereis_name/1`
`global:send/2`	`vnet:send/2`

Distribution primitives not listed here were not implemented for the virtual network, as they were either not required in our test cases, or could be sufficiently built by combining the supported primitives. For example, instead of `spawn(Node, Mod, Fun, Args)` it is possible to use `vnet:rpc(Node, erlang, spawn, [Mod, Fun, Args])` for spawning processes in a remote virtual node.

The `vnet` module also implements a name registry which can be used together with OTP behaviours relying on the `{via, vnet, Name}` naming scheme.

On top of implementing the necessary distribution primitives, the model needs to provide extra functionality for writing tests for distributed systems. Since node and network errors are crucial to our interest, APIs to stop and start virtual nodes, and to break and restore virtual network connections between individual pairs of nodes are required. Additionally, we wanted the model to work well together with Concuerror. This entails two more requirements: naming processes and minimizing the number of operations that can lead to races.

Naming processes is purely a cosmetic feature. Apart from their raw PIDs, Concuerror identifies processes with names like `<P.2.2.1.1>` that encode the order in which processes were spawned, but are not easily readable by humans. However, if a process has a registered name (using Erlang/OTP's native process name registry, not a custom name registry module like `global` or `vnet`) Concuerror will also show that name. Therefore, `vnet`'s name registry uses the native registry as a back-end for any processes that get registered, and also to identify helper processes used by `vnet` itself.

Making the library efficient for testing with Concuerror is a critical but not trivial problem. Each Erlang operation that may cause race conditions is logged by Concuerror as an event and, in Concuerror's current implementation, the computational cost for detecting such race conditions is quadratic in the number of such events. As a result, operations that would be considered trivial, such as looking up a PID by name, need to be optimized to minimize the events they generate, e.g., in this particular case, by storing the PID in a local variable instead of repeatedly using the `whereis` built-in operation. Moreover, `vnet` needs to avoid using operations that race with each other or with events generated by the test code itself.

3.2 Implementation

This section describes the most significant aspects of the actual implementation of the `vnet` library. Using the library, one can orchestrate a virtual cluster by starting and controlling virtual nodes, and spawning processes in the virtual nodes that are allowed to interact with each other. Note that the library does not control the entire Erlang node and there are native processes that are not related to the model. This category includes, for example, all processes in the supervision tree of the `kernel` application.

The functionalities provided by the `vnet` library are based on three kinds of system processes: *vnode*, *proxy*, and *connection*. They interact to simulate the distributed behaviour of Erlang, allowing users to have control over the whole virtual cluster. In the rest of this section, we describe how these processes work.

The vnode processes For each virtual node (vnode) in the virtual network, there is one corresponding *vnode* process. This process keeps track of the state of the vnode (whether it is started or stopped) and coordinates the transition between these states via a synchronous API. Its secondary role is to serve as a group leader for processes running in the vnode. The group leader of a process can then be used to determine in which vnode the process is running in. As the group leader is inherited by child processes upon spawning, it is an ideal candidate to serve transparently as the vnode marker. This mechanism is the same as the one used by Erlang/OTP to keep track of processes belonging to the same application.

Vnode processes are used by `vnet:rpc/4` to spawn a new process in the vnode to execute the request. Both native processes and processes running in vnodes may use `vnet:rpc/4`. This is the only mechanism provided by the `vnet` library for bootstrapping the system and starting initial processes in vnodes.

The proxy processes When using the `vnet` library, a process a in vnode $n1$ that would like to interact with (send a message to, link, or monitor) a process b in vnode $n2$, interacts instead with a *proxy process* created by the `vnet` library There are two reasons for this arrangement: the message ordering guarantees Erlang provides in a distributed setup and the handling of exit signals.

Proxy processes help to achieve the message ordering guarantees of a real distributed Erlang system (described in Section 2.1), because the extra hop introduced by sending messages via proxy processes identify the process pairs, and precisely targets inter-node messaging.

Regarding exit signals, observe that when the network connection goes down between the nodes, remote processes should be seen as terminated by the monitoring processes. However, since the model is simulating just a network error, the actual processes must not be terminated, so any native monitors and links to the actual processes would not be triggered. By placing monitors and links on a proxy process instead, it is possible to only terminate the proxy, without stopping the actual processes in the disconnected vnode.

Network connections can break independently between pairs of nodes, implying that a proxy process shall be specific to a pair of vnodes. Hence, in our example process b would need different proxies towards processes in vnodes $n1$ and $n3$: $p_{<b,n1\to n2>}$ and $p_{<b,n3\to n2>}$ respectively. Moreover, for reasons that will become clear in the first paragraph of Section 3.3, there is a separate proxy for each direction of communication, i.e., the proxy for messages from process a to b is different than the proxy for messages from b to a.

In our model, process a never sees the real PID of b and always sees the PID of its proxy ($p_{<b,n1\to n2>}$) instead. Whenever a sends a message to $p_{<b,n1\to n2>}$, the proxy will forward the message to the real b process. The proxy $p_{<b,n1\to n2>}$ will also link to b, so linking to/monitoring the proxy is equivalent to linking to/monitoring b itself.

Ensuring that processes of one vnode never see PIDs of processes of other vnodes is challenging, when arbitrary data can be exchanged between all processes via message passing. However, since all inter-vnode messages in the `vnet` library are going through proxy processes, it is the responsibility of these proxies to scan the messages for PIDs and translate them according to the following rules:

- Native processes' PIDs (not belonging to a vnode) are left intact.

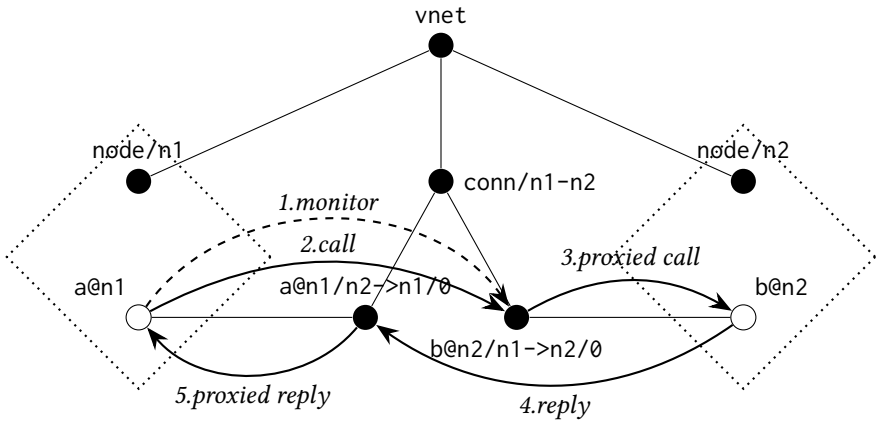

Figure 1. A `gen_server` call across vnodes using the distributed Erlang model. Dotted diamonds show the conceptual vnode boundaries. White circles correspond to 'user' processes and black circles to system processes of `vnet`. Links are shown with solid straight lines, monitors with curved dashed arrows (pointed at the monitored process), and messages with curved arrows.

- Real PIDs of processes running in a vnode (belonging to the sender's vnode) are replaced by their proxy's PID.
- Proxy processes' PIDs are replaced by the real PID of the proxied process, if the proxied process belongs to the recipient's vnode.
- Proxy processes' PIDs are replaced by the pid of a different proxy, if the proxy belongs to a different vnode than the recipient's, as proxies are specific to a pair of vnodes.

One more implementation detail worth mentioning is the use of an ETS table as a PID-type cache. Translating PIDs in proxy processes is a very common operation, which requires identifying the type of the process by looking up its group leader and process dictionary and (for non-proxy processes) the process dictionary of the group leader as well. To reduce the number of events Concuerror has to deal with, the result of these PID-type lookups is cached in an ETS table.

The connection processes For each connection between a pair of vnodes there is one corresponding connection process. This process keeps track of the connection's state (whether it is up or down) and maintains the list of proxy processes specific to this connection.

When a proxy process is translating PIDs in a message, it queries the connection process for the proxy's PID or requests it to create a new proxy when one does not exist yet. This way proxies are created on-demand, which helps keeping the number of processes used for modelling the network low.

Example Figure 1 shows the role of `vnet`'s system processes in performing a `gen_server` call across vnodes. All processes are labelled with their actual registered names.

Process *a* running in the *n*1 vnode execute a `gen_server` call towards process *b* running in *n*2 vnode. Gen_server calls start with monitoring the server, therefore a monitor will be placed on the respective proxy process. A message is then sent to the proxy, which forwards it to *b* in *n*2. When a reply is sent back, the respective proxy process for *a* in *n*1 is used.

When used as a name registry for user processes, the `vnet` module calculates a name using the name of the vnode as a suffix. For example the `gen_server` process was started with the name `{via, vnet, {n2, b}}`, and the `vnet` library turned that into `b@n2`.

Names of proxy processes contain the name of the proxied process and the name of the vnodes in the connection they are specific to. They also have an integer suffix, which is the number of times the connection has been lost before. For example, bringing down the connection between *n*1 and *n*2 would kill the `b@n2/n1->n2/0` proxy process; if the connection is later restored, a new proxy process will be created for the same `b@n2` process, named `b@n2/n1->n2/1` to distinguish it from its predecessor.

The vnode and connection processes (named `node/n1`, `node/n2` and `conn/n1-n2`) are not directly involved in performing the `gen_server` call. They are shown on the figure because they act as the parents of the previously discussed processes. These processes do have an active role in simulating connection or node failures however.

Finally, the top level process called `vnet` is a simple supervisor for all vnode and connection processes, hence we do not discuss it further.

3.3 Limitations

Proxied messages and monitors/signals The most significant limitation in our model is visible already in Figure 1: the reply message arrives to process *a* via a different proxy

process than the proxy that is monitored by process a. This means that if process b terminates after sending a reply there is a race condition between the delivery of the reply and DOWN messages. In other words, the vnet library cannot keep the guarantee that DOWN messages and exit signals arrive strictly after the last message that was sent by the dying process.

To guarantee correct ordering of reply and DOWN messages the reply message from process b to a should be sent via the same proxy process, p_b, that a used in place of b for monitoring and message sending. There are two ways for achieving this: either p_b could serve as a proxy for b too when it wants to interact with a, or b uses a different proxy, p_a, that is somehow aware of p_b and forwards the response to a via p_b. These alternatives are illustrated in Figure 2.

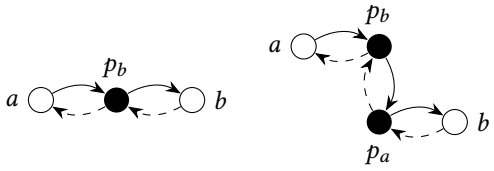

Figure 2. The two possible configurations for guaranteeing correct ordering of DOWN messages and exit signals. The path of a message from a to b is shown with solid arrows, and the path of a message from b to a with dashed arrows.

The first alternative cannot work, because in this scenario the proxy may receive messages from both a and b. But since the sender of Erlang messages is unknown, it would not know to which direction shall it proxy the messages. Information regarding senders could be added to messages by using a special send operation, but then one would have to consistently use such a special send operation, instead of the native one.

In the second alternative, proxy processes need to deal with two kind of messages: those routed to them via a different proxy for delivery and ordinary messages from processes in a vnode. The inter-proxy messages may come wrapped into a recognizable envelope that could also contain the PID of the final recipient. Ordinary messages would have to be routed to a different proxy. However, not knowing the sender of the ordinary message could still cause problems in this step. For example, how would p_a know that it received a message from b, and therefore it needs to route the message to p_b? If the message could come from a third process (c) in the same vnode as b, it would have to be routed to a different proxy process (p_c) instead.

The only solution for this problem would be to make the proxy processes dedicated to a pair of processes, not a pair of vnodes. A proxy receiving an ordinary message could then be sure which process this message came from. Unfortunately, this solution also cannot work, as it would also mean two processes in the same vnode would see the same remote process through different proxies, and therefore through different PIDs. That would mean PIDs in messages sent within the same vnode would need translating and therefore proxying too (as would PIDs stored e.g., in ETS tables, something possible only by replacing the ets module with a customised version performing the translation). Proxying between processes in the same vnode is also not possible if we want to use native process spawning, as spawn would also have to be intercepted to return the PID of the proxy, not the real process.

The conclusion is that guaranteeing the proper ordering of DOWN messages and exit signals with regard to ordinary messages is not possible in our model based on proxy processes. However, this should not be a show-stopper since there are workarounds to overcome this problem. Processes that communicate with a remote peer should handle DOWN messages or exit signals and discard any further messages sent by that particular peer. Figure 3 demonstrates this pattern applied to a gen_server.

```
handle_info({Pid, Msg},
            %% Verify with pattern matching
            %% that the message is coming from
            %% our peer
            S0 = #{pid := Pid}) ->
  S1 = on_msg(Msg, S0),
  {noreply, S1};
handle_info({'DOWN', Mon, process, Pid, Rsn},
            S0 = #{pid := Pid, mon := Mon}) ->
  S1 = on_down(Rsn, S0),
  %% Erase the process (and monitor) from the
  %% state; any late messages will be
  %% discarded as a consequence
  S2 = maps:without([pid, mon], S1),
  {noreply, S2};
handle_info(_Msg, S) ->
  %% Discard unexpected messages
  {noreply, S}.
```

Figure 3. Code snippet for a gen_server discarding messages arriving from its remote peer after the DOWN message.

Another type of processes that is vulnerable to this race condition is one-off workers that send their result back to their parent immediately before terminating. A workaround for this case is to use the exit signal for passing the result to the parent. Figure 4 illustrates how to transform a one-off remote worker pattern to prevent this race condition. The library's own vnet:rpc/4 primitive is implemented using this pattern.

PID visibility The second important limitation is that the vnet library relies on the assumption that processes in a

```
unsafe_remote_worker(Node) ->
  Parent = self(),
  Fun = fun () ->
            receive start -> ok end,
            %% Delivery of the message races
            %% with the delivery of the exit
            %% signal
            Parent ! {ok, do_work()}
        end,
  Pid = vnet:rpc(Node, erlang, spawn, [Fun]),
  Mon = monitor(process, Pid),
  Pid ! start,
  receive
    {ok, Result} ->
      demonitor(Mon, [flush]),
      {ok, Result};
    {'DOWN', Mon, process, Pid, Reason} ->
      {error, Reason}
  end.
```

```
safe_remote_worker(Node) ->

  Fun = fun () ->
            receive start -> ok end,
            %% Deliver the result via the
            %% exit signal, without race
            %% condition
            exit({ok, do_work()})
        end,
  Pid = vnet:rpc(Node, erlang, spawn, [Fun]),
  Mon = monitor(process, Pid),
  Pid ! start,
  receive
    {'DOWN', Mon, process, Pid,
     {ok, Result}} ->
      {ok, Result};
    {'DOWN', Mon, process, Pid, Reason} ->
      {error, Reason}
  end.
```

Figure 4. Unsafe and safe patterns for workers delivering the result of a computation to their parents before terminating.

vnode are not able to access the PID of processes in different vnodes, just the PID of their proxies. The message translation done by proxies is meant to cover PIDs sent to a process from remote processes, but messages are not the only way a process may acquire a PID. However, by paying more attention to the following scenarios when using the library one can ease the situation:

- Native processes can see the real PIDs of processes in vnodes, and can also communicate with them without going through a proxy. Therefore, native processes shall never send a PID to a process in a vnode.
- Some built-in operations are 'leaking' information, e.g., it is possible for processes in a vnode to list all processes in the Erlang node with `erlang:processes/0` and arbitrarily communicate with any of them, even though some would conceptually belong to different vnodes.
- A PID may be hidden from the translation code if converted into its string representation (from which it can be restored with `erlang:list_to_pid/1`). PIDs within variables of anonymous functions sent in the message cannot be translated either.

Other limitations There are a few more incompatibilities/limitations in the model, but they are less likely to surface in tests, either because these limitations are rather intuitive, or because they are related to features rarely needed in the kind of tests the `vnet` library is designed for.

To begin with, after disconnecting and reconnecting two vnodes the visible PID of remote processes will change, since a new proxy process will be started for them. This could cause problems if a process keeps the remote PID and try to keep sending messages to it after the reconnection.

Processes running in a vnode can interact with native processes that are not aware of our model's semantics, but this may cause trouble. Starting an OTP application in a vnode, for example, is not possible, as both the `vnet` library and the `application_controller` process would like to use the group leader of the application's processes for their own purposes. Instead, we suggest starting the top level supervisor.

Last but not least, observe that in a real system there are two distinct service layers controlling how a pair of nodes get connected: the physical network and the Erlang VM. For a pair of nodes to successfully connect, both layers need to work properly. Our model's `vnet:[dis]connect/2` operations treat these as a single, unified layer.

4 Case Studies

In this section we describe two case studies for the `vnet` library: the counter server and the KRED/KDB model. The counter server's purpose is to illustrate the main library primitives and give basic examples of its use in modelling. The KRED/KDB model is a case study from real-life, in which we used the library to verify the new architecture we are planning to introduce in KRED. We evaluate both case studies in Section 5.

4.1 Case Study: Counter Server

The counter server is a simple distributed system model that we built using the `vnet` library. The source code for all

components is available in the library's repository [1]. The system consists of a counter server node and a number of client nodes interacting with the server.

The counter server hosts a process that is a supervised, locally registered gen_server, which maintains a counter in an ETS table belonging to the supervisor. In this way, the counter can survive the crash of the gen_server process. The gen_server is implemented using the 'let it crash' philosophy, with any unknown requests leading to abnormal termination.

The clients can send requests to increment the counter, getting an integer number as a reply. Clients expect that the values they get are unique and strictly monotonically increasing. In our scenarios we have two different clients: those that use the server's API properly (i.e., send only increment requests) and those that send invalid requests. Each client is deployed to a separate node.

To run the distributed system within an Erlang node, the name registry and the virtual network of the `vnet` library is used. Names are registered via the `vnet` module. Observe that this is the only difference required to migrate OTP compliant processes to the `vnet` library: when providing the name of the process `{via, vnet, Name}` should be given instead of `{local, Name}`.

Figure 5 shows how to start a virtual network with one vnode called `server` and then start the supervisor of the counter server on the `server` vnode. The additional process created by the call to `erlang:spawn/3` is used to decouple the linked supervisor from the RPC worker which will exit abnormally (see Section 3.3).

```
{ok, _VnetSup} = vnet:start_link([server]),
RPC =
  [counter_server_supervisor, start_link, []],
vnet:rpc(server, erlang, spawn, RPC),
```

Figure 5. Starting the counter server on the server vnode.

There are two ways to communicate with the counter server running on the `server` vnode: either directly, using distributed Erlang message passing, or by using RPC calls. Both variants are shown in Figure 6.

```
vnet:rpc(server, counter_server, request,
         [{via, vnet, ?SERVER}, increment]).

counter_server:request(
  {via, vnet, {server, ?SERVER}}, increment).
```

Figure 6. Two variants for sending requests to the counter server.

In Section 5.1.1 we present test scenarios for this system, together with details about their execution with Concuerror.

[1] https://github.com/klarna/vnet

4.2 Case Study: KRED/KDB Model

The second case study is a model for a distributed database system that matches the new architecture of KDB, the database system used by KRED. The database should replicate data on multiple Erlang nodes. The system should also maintain a 'single source of truth' by electing a leader node, using an external write-ahead log service. We remind that the described architecture is the result of a redesign effort and can arguably be improved and simplified further, but this is beyond the scope of this work. We are merely interested in testing and proving correctness properties of this particular system.

We first describe the behaviour of the external write-ahead log service, followed by that of the Erlang database nodes.

4.2.1 External Write-Ahead Log Service

One of the main goals of the redesign was to replace the custom-developed replication logic of the database with an off-the-shelf solution. Apache Kafka [17] is a distributed streaming platform, which can be used to reliably transfer data between systems. Erlang's message passing can also be used for this purpose, but Kafka can additionally store the stream of transferred data in an ordered, fault-tolerant, and durable way, allowing easier recovery in case of a database node failure. In order to test the design of the new system, a model for the behaviour of Apache Kafka was therefore needed. We modelled only the features needed for our scenarios.

An Apache Kafka cluster manages a number of *topics*, each corresponding to a particular data stream. Other systems can publish and subscribe to any number of these topics. Each topic corresponds to a log of messages, stored in a (configurable) number of *partitions* in the Kafka cluster. When a new message is published to a topic, the cluster stores it in one of the topic's partitions and forwards it to all subscribers of the topic. Order is preserved between messages that end up in the same partition. Clients can also query about the total number of messages in a topic and ask for a particular message using the message's *offset*. In our use case, the database nodes use a single Kafka topic with a single partition.

In our model, the Kafka cluster corresponds to a single Erlang gen_server process. This process is not running inside a vnode, as the Kafka cluster is an external service. Clients publish a message to a topic by a call to the server, who replies with the offset in which the message was stored. After a client subscribes to a topic (also using a gen_server call) they will receive messages from the server every time a new message is published. Clients can also fetch particular messages, or ask for the total number of messages in a topic.

4.2.2 Erlang Database Nodes

The part of the system using the `vnet` library is the Erlang database nodes. Here we briefly discuss how transaction replication and leader election work, and highlight what kind of scenarios we want to test.

The KRED cluster is a fully connected cluster of Erlang nodes. Nodes are identical: each holds a local copy of the database, and is connected to the Kafka cluster. Nodes can join and leave the cluster dynamically. The cluster has two states: stable and unstable state. Read-only transactions are allowed in both states, but modifying transactions are allowed only when the cluster is in the stable state.

The cluster is in a stable state when there is exactly one *leader* node and zero or more *follower* nodes. The leader node holds the truth and all transactions are committed on that node. Transactions can be initiated on the follower nodes as well, but then they will actually be executed on the leader node, using an RPC. When a transaction arrives, the leader first locks all data that are accessed and modified by the transaction, preparing it to be committed. Following that, it publishes the transaction to Kafka. Once this is completed, the leader commits the transaction to its local database, and then notifies the process that has requested the transaction. Follower nodes are subscribed to Kafka, and follow the leader's messages to keep their database up-to-date. The database of the follower nodes are always almost up-to-date, it is naturally lagging behind but it is always consistent, since transactions are imported atomically.

Follower nodes monitor the leader node and can therefore notice when the leader is no longer accessible. When the cluster has lost its leader node it transits into the unstable state. The goal of the cluster is to leave this state, which requires the election of a new leader. All follower nodes keep reading messages from Kafka and once they have reached the end of the topic, their local database is identical to the lost leader thus they are qualified to enter the leader election as *candidates*. KRED nodes behave greedily, nominating themselves as leaders as soon as possible. Leader election happens on the same Kafka topic where the transactions are published. The algorithm relies on the causal ordering guarantees of Kafka, which serialises the candidates racing for the leader role. The algorithm uses a straightforward 'first write wins' strategy to decide the leader (i.e., the first node that publishes their candidacy is elected). The leader is announced on the topic as well. When a new leader has been elected the rest of the cluster starts following the new leader. When the followers have managed to connect and monitor the new leader the cluster can leave the unstable state. Any failure in this sequence of steps will make the nodes restart the leader election.

It is also worth reminding that the leader may not really be down, just isolated from the rest of cluster, hence followers will *consider* it down. In this case, too, a new election will begin and the leader needs to stop accepting transactions and enter the election with the rest of the cluster. Observe that if the cluster is in stable state, it is only the leader who publishes messages to Kafka. Therefore, the leader can notice that an election has begun if it maintains the last offset of its last write and checks for gaps upon publishing.

The tests we executed for this model are described in respective evaluation section (5.1.2).

5 Evaluation

In this section, we present an evaluation of our distributed Erlang model. Our question is whether the `vnet` library is powerful enough to enable checking properties of a distributed system. In other words, can we write interesting models and scenarios using the library and are those models expressive enough to test meaningful properties. If so, then our distributed Erlang model enables testing and verifying distributed systems with tools developed for a single node tests.

5.1 Scenarios and Properties

We start by showing the test scenarios we implemented using our distributed Erlang model for both case studies. For the counter server model, all scenarios are described in detail and the code is available in the `test` directory of the `vnet` library's repository[2]. For the KRED/KDB model, we only give high level descriptions of some scenarios and indicative results, respecting Klarna's intellectual property policies.

All described scenarios were given as inputs to Concuerror[3] runs, on a MacBook Air laptop (13-inch, Early 2015, 1,6 GHz Intel Core i5, 8 GB 1600 MHz DDR3), with Erlang/OTP 20.3 built using the kerl tool[4]. All runs were verification attempts, using the optimal DPOR algorithm, with further optimizations for Erlang's message-passing-related race conditions [1, 5] (`--dpor optimal, --use_receive_patterns` options), exploring all possible interleavings (no schedule bounding [3]), continuing exploration after errors were found (`--keep_going` option), up to a threshold of 99999 total interleavings (`--interleaving_bound` option).

5.1.1 Counter Server

Table 2 shows results for test scenarios for the counter server system (Section 4.1). Each scenario has two variants: one using distributed Erlang message passing directly (using proxy processes and labelled as `_proxy`) and one using RPC calls (labelled as `_rpc`). The names shown in the table correspond to `_scenario` functions within the module called `counter_server_example`.

One client sending two valid requests (`twice_valid`)
In this scenario we start the virtual network that consists

[2] https://github.com/klarna/vnet
[3] Version used: https://github.com/parapluu/Concuerror/tree/3cc746
[4] https://github.com/kerl/kerl

Table 2. Results from evaluating scenarios of the counter server model using Concuerror. Tests marked with an asterisk (*) reached our interleaving threshold.

Name	Errors	Total	Time (s)
twice_valid_proxy	0	10	4
invalid_proxy	0	4012	77
invalid_same_gen_proxy	136	4012	82
disconnect_proxy	0	1150	13
node_down_proxy *	664	99999	1300
twice_valid_rpc	0	26	4
invalid_rpc	0	3350	73
invalid_same_gen_rpc	312	3350	98
disconnect_rpc	0	453	8
node_down_rpc	0	54722	721

of two vnodes: the server and the client. The client sends two requests to the counter server. The goal of the test is to check that the client sees monotonically increasing values.

Two clients sending valid & invalid requests (`invalid`) In this scenario we start the virtual network that consists of three vnodes: the server, the good client and the bad client. The good client sends two valid requests in parallel with that the bad client sends an invalid request that crashes the server. We check that if the good client receives valid replies then the counter values are monotonically increasing.

We also tried a variant of this scenario (referred to as `invalid_same_gen`) containing an additional, incorrect assertion: the generation of the server is the same in both requests. Clearly, if the server crashes and restarts between the requests this property does not hold.

Vnodes are disconnected during the test (`disconnect`) In this scenario the virtual network is started with a server and a client vnode. The client will send two valid requests to the server while the connection between the server and the client is broken. Since Concuerror executes the test ensuring that all possible schedulings are explored, we can be sure that the scenario when the client is calling the server while the connection is broken between the nodes is exercised. We assert that the counter is increased monotonically and the generation of the server remains the same.

Vnode crashes during the test (`node_down`) In this scenario the virtual network is started with a server and a client vnode. The client will send two valid requests to the server while the server vnode is crashing. Here we expect that while the client can send requests to the server all responses will respect the constraint on the counter value and the server generation does not change. Of course, there will be schedulings when the client cannot complete two requests because the server goes down earlier. Observe that the server will not come back, so once it is down it will stay down.

The distributed message-passing scenario reaches the interleaving threshold finding errors. Alas, as is often the case, those errors were in the specification. Namely, when writing the specification of error replies we forgot to include the case in which the sever dies during serving a valid request, which is clearly possible in this situation, if the vnode goes down. Meanwhile, the RPC variant of this scenario terminates without errors, as the `badrpc` error reason was (correctly) specified as an expected one.

5.1.2 KRED/KDB Model

It is easy to see, from the description given in Section 4.2, that several choices can be made when writing scenarios for the KRED/KDB model, such as how many nodes there will be in the cluster, when will each node be started and connected to the rest, whether any transactions will be initiated or not and on which node will they be initiated from, and finally if and when any nodes will be killed or disconnected. There exist also several correctness properties one can be interested in, such as whether a leader node is always eventually elected, whether any nodes are crashing (expectedly or unexpectedly), and whether transactions always reach an acceptable final state.

We report indicative results from three test scenarios to illustrate the capabilities of our model.

Scenario with a transaction The first scenario is a replication scenario.

We start two vnodes, start the database application on both of them and begin executing a transaction on one of the vnodes. If the vnode has not completed the leader election process before the transaction is submitted, the transaction will fail. We then wait for the system to reach a stable state and check that the outcome is one of several valid ones: e.g., no vnodes crashed, and the transaction is either committed everywhere or failed everywhere.

This scenario leads to 18672 schedulings, explored in approximately 7 minutes. The slower exploration rate compared with the `node_down_rpc` scenario in Table 2 (45 versus 76 interleavings/s) is due to the increased number of events in the scenario.

Restricted scenario with a transaction By constraining a number of parameters in the previous scenario, we can get a much smaller search space. Rather than starting everything at the same time, we start a vnode, start the database application, and at the same time start a transaction on the same vnode. We then wait for the transaction to complete or be discarded and then start a second vnode, start the database application there too, and then wait for the system to reach a stable state. As safety properties, we check at the end of the execution that the final leader is always the first vnode, no vnodes have crashed, there are no transactions at

a pending state on any vnode, and the local database copies of the vnodes are consistent with each other. As a reachability property, we check that the transaction is sometimes successful.

This restricted scenario has only 172 possible schedulings, explored in 7s.

Scenario with a node failure The last scenario is a failover test, in which we check that the cluster is able to recover when the leader is lost.

We start a vnode and start the database application on it, then wait for leader election to be completed, then start a second vnode and the database application on that vnode too and wait for the vnode to join the cluster. We then stop the first vnode and wait for the system to reach a stable state.

We then check that two leaders were elected during the test, there is exactly one crashed vnode (the first vnode), and the cluster has a leader in the end (the second vnode).

This scenario had 288 schedulings, explored in 9s.

5.2 Lessons Learnt

We observed that even though the `vnet` library supports supervisors and gen_* OTP behaviours, they lead to state space explosion in Concuerror. When a child of a supervisor terminates, a number of signals, messages arrive to different processes. These messages can race with each other in some extent, hence the state space grows significantly. Our take away from this observation is that when using the `vnet` library to build models that will be tested by Concuerror, the number of OTP behaviours should be kept to the meaningful minimum to have tests executable in a reasonable time. This is the reason for the timeout reported in Table 2, but further improvements in the `vnet` library and Concuerror can mitigate this issue.

A more important lesson we learnt while working with Concuerror is to not write scenarios that do 'everything at once', as this can lead to impractically large number of schedulings (e.g., compare the difference between the scenario with a transaction and its restricted version). Once we understood this and kept properties separated, we could verify the correctness of our scenarios in reasonable time.

5.3 Overall Assessment

As Sections 5.1.1 and 5.1.2 show, we were able to use the `vnet` library to model simple and more advanced distributed systems. Virtual nodes can be started, stopped using a virtual network where failures are easily injectable. The virtual nodes impose a few restrictions in the code, however, they do not restrict the user what to run on them, even supervisor trees are assignable to a virtual node. Thus, complex models can be built on top of the library.

We observed that our distributed Erlang model was powerful enough to support all test cases to implement. It did not constrain us, instead, it was an enabler to use Concuerror to run cluster tests in a single Erlang VM. Combining Concuerror with the `vnet` library gives a powerful verification tool in the hand of the users. Safety properties, liveness properties are observable and testable. Since the model is runnable, long-running, critical projects can be started using these two tools as a basis. Once a model of the distributed system has been developed and its properties have been tested, the model can serve as a base for development in production.

6 Related Work

Testing distributed systems for errors caused by node and connection failures is a topic of broader interest. Approaches that do not require modelling (e.g., Jepsen [15]) have been successful, but still require expert knowledge of the underlying systems [2]. Moreover, testing techniques can never verify that a particular scenario will always behave correctly, as they cannot systematically explore *all* possible behaviours (they can at best offer probabilistic guarantees [8, 18]). In this work we present an approach that is also suitable for verification.

Concuerror has been previously shown to be useful for using a test-driven development approach when working on single-node Erlang applications [13]. There has also been prior work on modelling a particular distributed system (a CORFU cluster [7]) using Erlang processes, and then testing and verifying it with Concuerror [4]. In this work, we combine and build upon these experiences, by describing a general model for distributed Erlang systems, that can be used for testing and verification of distributed systems.

A formal model for the communication between Erlang nodes has been described in a proposed formal specification of Erlang [16]. From that work, our model reuses the idea of a 'node controller', but separates its functionality in smaller modules.

7 Conclusion

The work presented in this paper is just the beginning of our journey, rolling out changes in production is awaiting for us. Modelling enforced us to decouple concepts, focus only on the relevant aspects. Using the distributed Erlang model allowed us to verify early our distributed algorithm using a tool capable of testing inside a single node. The tests we ran increased confidence, showing that desirable properties of the system hold.

In this paper we presented a broadly usable model of distributed Erlang system that is executable in a single VM. We demonstrated the capabilities of the model using two case studies. The authors believe that the model can be an enabler for other projects as well, since it unlocks testing options available only inside a single Erlang node. Therefore, changes in the core part of large-scale industrial Erlang applications can be rolled out with increased confidence.

Acknowledgments

The authors thank the anonymous reviewers for their comments and suggestions that helped significantly in improving this paper.

References

[1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2017. Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. *J. ACM* 64, 4, Article 25 (Aug. 2017), 25:1–25:49 pages. https://doi.org/10.1145/3073408

[2] Peter Alvaro and Severine Tymon. 2017. Abstracting the Geniuses Away from Failure Testing. *Communications ACM* 61, 1 (Dec. 2017), 54–61. https://doi.org/10.1145/3152483

[3] Stavros Aronis. 2018. *Effective Techniques for Stateless Model Checking*. Ph.D. Dissertation. Uppsala University. http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-333541

[4] Stavros Aronis, Scott Lystig Fritchie, and Konstantinos Sagonas. 2017. Testing and Verifying Chain Repair Methods for CORFU Using Stateless Model Checking. In *Proceedings of the 13th International Conference on Integrated Formal Methods (IFM 2017)*. Springer, Cham, 227–242. https://doi.org/10.1007/978-3-319-66845-1_15

[5] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. 2018. Optimal Dynamic Partial Order Reduction with Observers. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2018)*. Springer, Cham, 229–248. https://doi.org/10.1007/978-3-319-89963-3_14

[6] Stavros Aronis and Konstantinos Sagonas. 2017. The Shared-memory Interferences of Erlang/OTP Built-ins. In *Proceedings of the 16th ACM SIGPLAN Workshop on Erlang (Erlang 2017)*. ACM, New York, NY, USA, 43–54. https://doi.org/10.1145/3123569.3123573

[7] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*. USENIX Association, San Jose, CA, 1–14. http://dl.acm.org/citation.cfm?id=2228298.2228300

[8] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/1736020.1736040

[9] Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. 2013. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation (ICST '13)*. IEEE Computer Society, 154–163. https://doi.org/10.1109/ICST.2013.50

[10] Koen Claessen and Hans Svensson. 2005. A Semantics for Distributed Erlang. In *Proceedings of the 4th ACM SIGPLAN Workshop on Erlang (Erlang '05)*. ACM, New York, NY, USA, 78–87. https://doi.org/10.1145/1088361.1088376

[11] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-order Reduction for Model Checking Software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315

[12] Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

[13] Alkis Gotovos, Maria Christakis, and Konstantinos Sagonas. 2011. Test-Driven Development of Concurrent Programs using Concuerror. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang (Erlang '11)*. ACM, New York, NY, USA, 51–61. https://doi.org/10.1145/2034654.2034664

[14] Fred Hebert. 2013. *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA. http://learnyousomeerlang.com

[15] Jepsen. 2018. Distributed systems safety research. http://jepsen.io. Accessed: 2018-06-14.

[16] Hans Svensson, Lars-Åke Fredlund, and Clara Benac Earle. 2010. A Unified Semantics for Future Erlang. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang (Erlang '10)*. ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/1863509.1863514

[17] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. 2015. Building a Replicated Logging System with Apache Kafka. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1654–1655. https://doi.org/10.14778/2824032.2824063

[18] Xinhao Yuan, Yang Junfeng, and Ronghui Gu. 2018. Partial Order Aware Concurrency Sampling. In *Computer Aided Verification (CAV 2018)*. Springer, Cham, 317–335. https://doi.org/10.1007/978-3-319-96142-2_20